[**漫谈WebSphere应用服务器之事务**](http://www.cnblogs.com/sunwei2012/archive/2010/01/08/1642295.html)

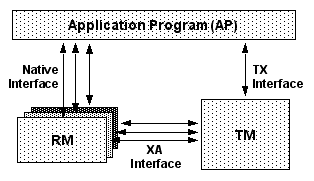
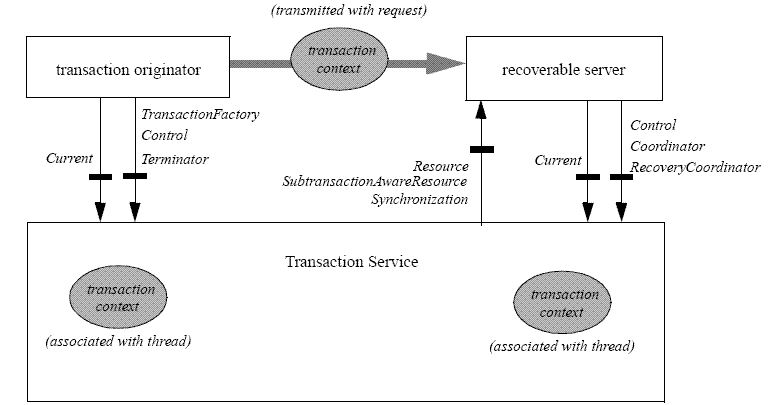
**开场白：**既然是漫谈，肯定是泛泛而谈。求多而不求精是目标，达到抛砖引玉、引发思考或者讨论乃最终目的。

    当前，企业应用变得越来越复杂，常常需要多个业务系统一同协作才能完成一个端到端的业务。因此如何协调多个业务系统来保证业务的事务性成为一个很关键问题。

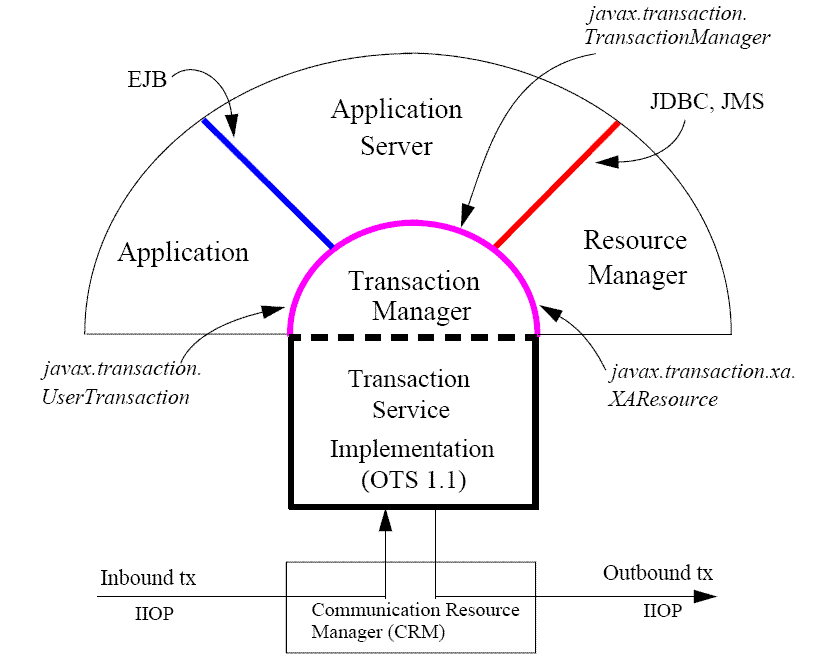
    WebSphere应用服务器（WAS）作为一个先进的J2EE平台，其本身也是一个事务管理器。它实现了JTA（Java Transaction API）和JTS（Java Transaction Service），对分布式事务提供了强大的支持。同时WAS还提供了有价值的编程模型扩展来最大限度的保证多个资源之间的事物一致性。

    当然了，既然定位是漫谈，肯定是泛泛而谈。求多而不求精是目标，达到抛砖引玉、引发思考或者讨论乃最终目的。所以本文会从事务的历史开始谈起，然后到X/Open的DTP模型，CORBA的OTS一直到J2EE的JTS；最后，既然该文要和WebSphere应用服务器扯上关系，当然得写上几笔WebSphere相关的东东。因为都是WebSphere特定的内容，如果对此不关心完全可以视而不见......

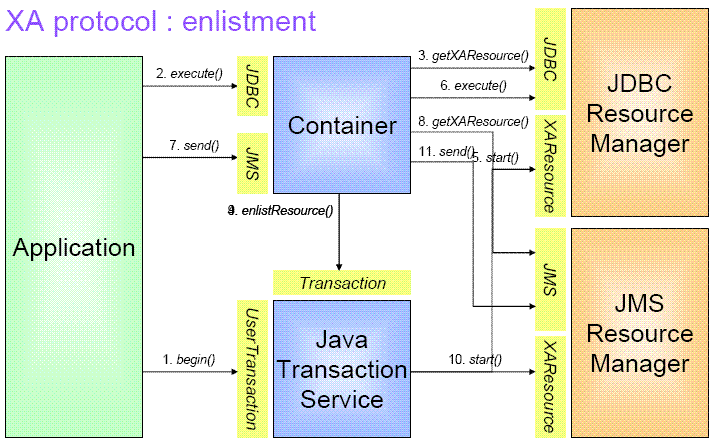
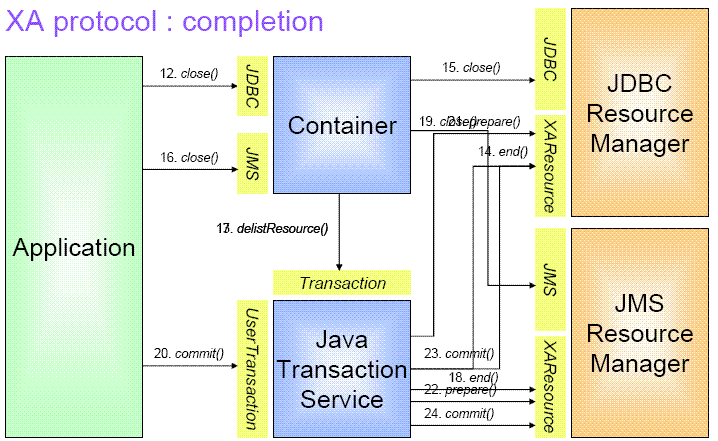
## 1.事务及事务管理器的基本概念和历史

   说起事务，大部分人都马上能说出事务的四个属性：ACID（原子性、一致性、隔离性和持久性）。  
   事务这个概念最先来自**数据库** 领域。我们可以看一下wikipedia上如何定义数据库事务的：  
A database transaction comprises a unit of work performed within a database management system (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions.  
  
从该定义其实已经可以体现出事务的四个特性了。而提供事务能力的事务处理系统早在60年代就出现了，典型代表有到现在还在企业中稳定运行的**IBM** 的CICS、BEA的Texedo等。而后事务处理最为一个新兴的学科开始发展起来，并在伟大的图灵奖得主、一个传奇式的人物 - Jim Gray下得到发扬光大。他的事务处理-概念和技术这本著作也成为这个领域的不朽经典。  
  
除此之外，影响深远的当属X/Open提出的DTP模型了。  
在 90年代，随着分布式计算的兴起，企业的计算模式也发生了质的变化。一个业务的生命周期往往需要跨越多个系统和多种资源（如数据库，消息系统等）。因此如 何保证这种分布式计算模式下面的事务性成了当时研究的热点。其中X/Open的DTP模型规范了事务处理系统中各组件的职能及交互的契约，它也成为目前这 个领域的事实标准。  
  
  
如上图所示，该模型提供了一个构建分布式事务处理系统的方法。其核心包括三个组件和两组接口。  
三个组件：  
   1. 应用程序（AP）：应用程序是事务的发起者；它决定了一个事务的生命周期（比如什么时候开始一个新的事务、什么时候该提交或者回滚事务）。  
   2. 事务管理器（TM）：如其名字所表述的，它提供了对事务的管理和恢复。  
   3. 资源管理器（RM）（Resource Manager），负责对资源的访问。  
    
两组接口：  
   1. TX接口。该接口定义了应用程序和事务管理器之间交互的方法。比如应用程序可以调用事务管理器的tx\_open/tx\_close方法用来建立/断开事务 管理器。然后使用tx\_begin/tx\_rollback/tx\_commit来通知事务管理器开始/ 回滚/提交事务。  
   当事务管理器接收到开始/回滚/提交事务的请求后，事务管理器必须告知加入到当前事务的资源管理器来执行对应的操作，这是通过另一组XA接口来实现的。  
  2.  XA接口定义了事务管理器和资源管理器之间交互的方法。重要的的方法有：  
       xa\_open/xa\_close。事务管理器通过调用这两个方法来通知资源管理器进行初始化和结束。  
      xa\_start/xa\_end。事务管理器通过这两个方法来通知资源管理器开始和结束事务。这两个方法也就确定了一个事务的边界。  
      xa\_prepare/xa\_commit/xa\_rollback。事务管理器通过两阶段提交协议来通知资源管理器进行事务的提交或者回滚。  
  
   在这个模型下，一个典型的事务处理可以表示如下：  
   -------------------------------------------  
  //1. 应用程序开始事务。  
    tx\_begin.  
    //应用程序进行业务操作，在需要保证事务的部分开始事务。  
    tx\_begin  
    //应用程序调用资源管理器进行业务操作  
         //事务管理器通知资源管理器进行初始化   
             xa\_open  
               //事务管理器通知资源管理器事务开始  
                 xa\_start  
                     
     //资源管理器记录事务性操作。。。  
   
     //最后应用提交事务。  
     tx\_commit  
        //事务管理器通知资源管理器开始两阶段提交，最后结束事务  
              xa\_prepare  
             xa\_commit  
            xa\_close  
    tx\_close  
--------------------------------------------------------------------  
  
   CORBA的兴起也给X/Open的DTP模型找到了一个绝好的实践场地。  
   在OMG的OTS规范中给出了如下的事务架构：  
可以看到该模型其实和DTP模型是完全一致的，只不过是用CORBA来重新描述了而已。而且在OTS中也明确说明了是可以和X/Open的DTP互操作的。

## 2.  J2EE事务

      网络就是计算机这个口号虽然没有造就一个互联网**软件** 帝国，却让Java成为了企业级应用的首选。随着EJB编程模型在J2EE中的引入，JTA和JTS也随之发布。  
      如下图所示，J2EE世界中的事务模型完全是延续DTP的。它也定义了应用、资源管理器和事务管理器。  
     JTA协议简单来说就是用Java来定义了X/Open的DTP模型中的两组接口（TX和XA）。具体而言，JTA把应用程序分为了两类，一类是真正的客 户应用程序，另一类是应用服务器。针对这两类事务客户端它分别定义了javax.transaction.UserTransaction和 javax.transaction.TransactionManager接口，这两组接口直接对应了TX接口。通过UserTransaction接口，客户应用程序就可以显示的控制事务的开始和结束。但在J2EE的世界中更推崇让容器来管理事务，这样应用无需硬编码，只要通过声明式的方式就可以让自己的EJB运行在特定的事务上下文中。在J2EE规范中定义了好多种类型的事务属性，根据这些属性容器在调用EJB之前和之后就会执行相应的事务控制操 作。比如如果事务属性设置为RequireNew：  
-----------------------------------------------------------------------  
          //在开始调用Bean方法之前，容器**代码** 先拿到一个事务管理器对象。如何拿到  
        //这个对象在规范中没有说明，因此不同的容器提供商有不同的方法。  
        TransactionManager txManager = TransactionManagerFactory.getTransactionManager();  
         
        //由于是需要新的事务，因此需要把当前已有的事务挂起（如果有事务的话）。  
        Transaction current = txManager.suspend();  
         
        //接下来就可以发起一个新的事务了。  
        txManager.begin();  
         
        //然后容器把请求发给EJB实例，业务逻辑开始执行。在执行过程中任何  
       //事务相关的操作都会被自动的加入到当前事务中。最后执行完成推出。  
         
        //容器重新接管控制权，如果没有异常，则提交当前事务。  
        txManager.commit();  
         
        //最后恢复被挂起的事务。  
        txManager.resume(current);  
----------------------------------------------------------------------  
而 XA接口则是由javax.transaction.xa.XAResource来提供的。该接口定义了资源管理器和事务管理器之间的契约。在上一章节中 我们已经 简要的介绍过XA接口的方法。这儿需要注意的是在J2EE的环境下，open和close并不会被调用到，因此资源的初始化和关闭时由资源管理器自己完成 的。  
  
而JTS相对而言就比较低层得多，它规范了一个J2EE下的事务管理器在上层需要提供对JTA的支持，而下层则需要实现OTS的Java映射。  
  
说到这儿，J2EE事务基本上可以告一段落。但其实还有一个问题是在上述的代码中，业务逻辑进行事务性操作的时候，比如往数据库里面加入一条记录，或者往 JMS Queue中写入一条消息。这些资源是如何自动的加入到当前的事务中呢？这部分内容是不可能在J2EE的规范中找到的。目前绝大部分容器的实现都是在XA 数据源上做的文章。比如我们在WebSphere应用服务器上定义了一个jdbc/Order的数据源，然后在一个Session Bean中用下面的示例代码来添加一个订单：  
  
---------------------------------------------------------  
InitialContext ic = new InitialContext();  
DataSource ds = (DataSource) ic.lookup("jdbc/Order");  
Connection con = ds.getConnection();  
Statement st  = con.createStatement();  
String **sql** ="insert into order .....";  
st.execute(sql);  
//最后cleanup  
st.close();  
con.close();  
  
---------------------------------------------------------  
在 上述例子中，如果这个Session Bean的事务标记为Required、RequiredNew、Support，则这儿的数据库操作会自动加入到当前的事务中。如何自动加入的秘密就在于这儿返回的数据源。一个简单的实现是当ds.getConnection方法被调用时，数据源的实现里面首先调用底层的资源管理器拿到一个 Connection，然后调用Connection的getXAResource拿到XAResource，接着拿到当前的 transactionManager应用，然后通过 transactionManager.getTranaction().enlistResource把该资源加到当前的事务中。最后返回一个 Connection的封装。通过这种方式资源就被自动的加入到当前事务中，然后在业务逻辑执行完成后，当容器重新拿到控制权后，容器就会根据配置来调用 事务管理器来完成事务的提交或者回滚。

## 3. WebSphere事务

  WAS作为一个J2EE服务器，自然也就实现了JTS跟JTA。这儿再温习一下整个事务的一个完整的运作过程.。这儿我们使用如下的一个场景：应用在完成 一个业务时需要同时访问数据库和消息系统，而且需要保证对这两种资源的更新的事务性，这儿应用使用了userTransaction来自己控制事务。示例 代码如下：  
  
-----------------------------------------------------     
        InitialContext ic = new InitialContext();  
        DataSource ds = (DataSource) ic.lookup("jdbc/order");  
        ConnectionFactory connectionFactory =  
            (ConnectionFactory)ic.lookup("jms/orderCF");  
  
        UserTransaction ut = (UserTransaction) ic.lookup("java:comp/UserTransaction");  
         
        try{  
             
            ut.begin();  
             
            javax.jms.Connection jmsConnection = connectionFactory.createConnection();  
            Session session = jmsConnection.createSession(true, Session.AUTO\_ACKNOWLEDGE);  
            Queue sendQueue = (Queue) ic.lookup("jms/orderQueue");  
            MessageProducer producer = session.createProducer(sendQueue);  
            jmsConnection.start();  
            TextMessage requestMessage = session.createTextMessage("**data** ");  
            producer.send(requestMessage);  
  
            session.close();  
            jmsConnection.close();  
             
             
            java.sql.Connection jdbcConnection = ds.getConnection();  
            java.sql.Statement st = jdbcConnection.createStatement();  
            String sql = "insert into ...";  
            st.execute(sql);  
             
            st.close();  
            jdbcConnection.close();  
             
            ut.commit();  
        }catch(Exception e) {  
            try {  
                ut.rollback();  
            } catch (IllegalStateException e1) {  
                // TODO Auto-generated catch block  
                e1.printStackTrace();  
            } catch (SecurityException e1) {  
                // TODO Auto-generated catch block  
                e1.printStackTrace();  
            } catch (SystemException e1) {  
                // TODO Auto-generated catch block  
                e1.printStackTrace();  
            }  
        }  
  
-----------------------------------------------------  
  
我们可以用下面的两幅图来分析整个事务的处理过程。  
首先看一下这两个资源是如何注册到事务管理器中的。  
  
  
  
1. 应用通过userTransaction.begin()开始事务。JTS把创建事务并把它关联到当前的线程上。  
2. 用户执行JDBC操作：拿到数据源、连接、创建Statement，最后调用statement.execute来执行sql语句来在数据库中创建一条新条目。  
3. 因为这儿使用的statement是经过容器包装的一个Wrapper，在其实现中首先向真正的XAConnection拿到XAResource。  
4 然后调用transactionManager.getTransaction().enlistResource()方法来把该XAResource加入到当前事务中。  
5  事务管理器调用该XAResource的start方法来通知资源管理器开始记录事务性操作。  
6 最后调用真正的Statement的execute()方法来执行数据库操作。  
  
7到11步和上述类似。通过类似的操作把JMS资源加到当前的事务中。  
  
当业务步骤完成后，调用close来关闭连接，然后调用commit来提交事务。这些操作都会触发对应的XA的动作，如下所示：  
  
  
12. 当connection的close方法被调用到后  
13. 容器提供的Connection wrapper会先调用transactionManager.getTransaction().delistResource()方法来停止记录。  
14. JTS调用XAResouce的end方法来结束事务。  
15. 真正的connection的close方法被调用，资源管理器释放资源。  
16 到19步完成JMS的close操作。  
20. 然后userTransaction的commit被调用，此时开始两阶段提交。  
21 首先JTS向JDBC的XAResource发起prepare请求，如果返回Prepared，则  
22 JTS向JMS的XAResource发起prepare请求，如果还是返回Prepared，则  
23. 调用JDBC的XAResource的commit方法落实所有修改  
24. 调用JMS的XAResource的commit方法落实所有修改  
  
除了经典的XA的支持，还值得提及的是WAS的一些特别的扩展。  
   
  首先要介绍的是本地事务  
  这儿先回顾一下为什么在EJB这种分布式计算模型下面为什么还有本地事务的需求。让我们先看一下在J2EE规范中明确提到了未指定的事务上下文这个概念。  
   
--------------------------------------------------------------------------  
  The term “an unspecified transaction context” is used in the EJB specification to refer to the cases in which the EJB architecture does not fully define the transaction semantics of an enterprise bean method execution:  
  
The execution of a method of an enterprise bean with container-managed transaction demarcation for which the value of the transaction attribute is NotSupported, Never, or Supports (onyl applies to the case when the client calls without a transaction context)  
  
The execution of the ejbCreate<METHOD>, ejbRemove, ejbPassivate, and ejbActivate methods of a session bean with container-managed transaction demarcation.  
  
The execution of the ejbCreate<METHOD> and ejbRemove methods of a message-driven  
bean with container-managed transaction demarcation.  
-----------------------------------------------------------------------  
  
在上述提到的场景中，由于缺乏事务的语义，因此没有办法规定在这些场景下的事务属性。因此规范索性把这部分定义留给了具体的容器提供商。  
在 这些情况下，WAS总是会启动一个本地事务（LTC）来对事务进行控制。LTC针对资源管理器的本地事务提供了三个层次的支持。首先是事务的边界，缺省是 Bean方法；然后是对于没有完成的本地事务的处理策略，缺省是回滚。最后是对资源的回收，保证在本地事务推出时候里面所使用到得资源得以释放。  
LTC本身没有编程接口，只能通过WAS对EJB部署描述符的扩展来配置。  
   
分布式事务的一些扩展  
      分布式事务上WAS也提供了两种扩展。  
      第一种是Last participant。这种扩展用在一个事务中如果有多个资源管理器支持XA（2PC，两阶段），但恰好有一个只能支持RMLT （resource manager local transactions）（1PC，一阶段，没有Prepare，只有commit）的场景下。在这种情况下， 可以通过配置last participant来支持这种场景。具体实现上是在开始两阶段的prepare时，事务管理器首先询问其他的XA资源是否准备好，如果回答都是OK， 接下来就调用1PC资源的提交，如果提交成功，则调用其他的2PC的commit，否则rollback。  
   
      上面的场景要求只能有一个1PC的资源，如果把这个限制放松，则没有办法用这个方式来实现对不同类型的资源的统一协调。ActivitySession就 是为了满足这种涉及了多个1PC的资源的应用场景。可以说它扩展了传统的XA的概念，把它延伸了道了1PC资源上。和分布式事务类似，它的事务边界的划分 也可以通过声明式或者是编程来实现。但不同点在于它没有Prepare阶段，也没有Recovery。  
       
其他编程接口  
      在J2EE规范中，UserTransaction只能在Bean或者Servlet里面使用。那么在普通的POJO里面，如果访问事务，如果发起新的事务呢？WebSphere提供了下面的一些办法：  
  
ExtendedJTATransaction.  
通过该接口可以知道当前有没有全局事务，如果有则可以注册一个SynchronizationCallback对象到当前的事务或者容器中所有的事务中。  
可以在EJB、Servlet、**JSP** 或者普通的POJO中通过下面的代码访问到该对象：

ExtendedJTATransact

ion

exJTA = (ExtendedJTATransact

ion

)ctx.lookup("

java:comp/websphere/ExtendedJTATransact

ion

");

//判断是否有活动的事务

if (exJTA.getGlobalId() != null) {

//给当前的事务注册一个callback

SynchronizationCallback sync = new SynchronizationCallback();

exJTA.

**registerSynchronizationCallbackForCurrentTran**

(sync);

}

ExtensionHelper.

如果需要在POJO中发起新的事务，则可以用ExtensionHelper。示例代码如下：

InitialContext ctx = new InitialContext();

ExtensionHelper eh = (ExtensionHelper) ctx.lookcup(ExtensionHelper.JNDI\_NAME);

TransactionControl tc = eh.getTransactionControl();

TxHandle txh = tc.preinvok

e

();

try {

// Do some work ...

// Commit the transaction

tc.postinvoke(txh);

} catch (Throwable fatalException) {

// Rollback the transaction

tc.handleException(txh);

}

UOWManager  
在WAS6.1以后提供了一个更简洁的接口来实现对UOW的管理。上面在POJO中启动一个事务的代码可以重写如下：

try

{

UOWManager uowManager = UOWManagerFactory.getUOWManager();

uowManager.runUnderUOW(UOWSynchronizationRegistry.UOW\_TYPE\_GLOBAL\_TRANSACTION, false, new UOWAction()

{

public void run() throws Exception

{

// Perform. transactional work here.

}

});

}

catch (UOWActionException uowae)

{

// Transactional work resulted in a checked exception being thrown.

// The UOW was not affected.

}

catch (RuntimeException re)

{

// Transactional work resulted in an unchecked exception being thrown.

// The UOW was rolled back

}

catch (UOWException uowe)

{

// The completion of the UOW failed unexpectedly.

}

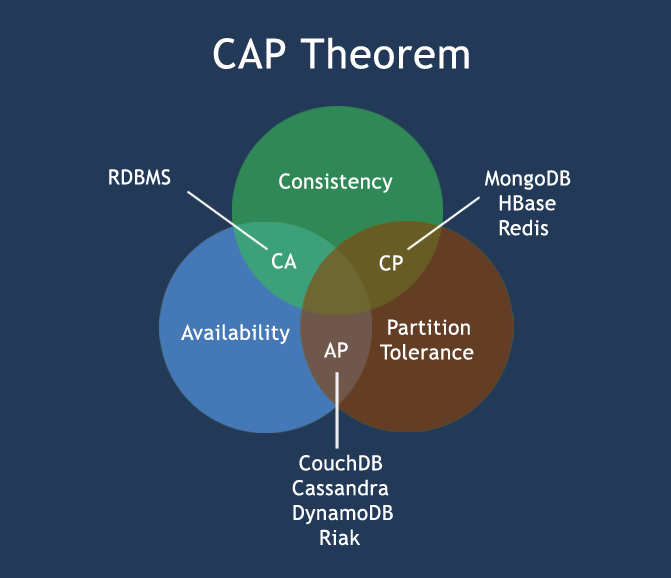
## 4. ****Web**** 服务事务

现在SOA火的一塌糊涂。而作为SOA基石之一的Web服务业早就深入人心。那么在处处都是服务的时候，如何保证服务之间交互的事务性也是必须要解决的问题。  
和 CORBA的OTS、J2EEE的JTA/JTS一样，Web服务也搞了一套自己的事务解决方案。目前，Web服务的事务是由WS- Coordination和WS-Transaction两个规范构成。WS-Coordination中提出了一套WS-Coordination Framework来定义了在Web服务的场景下如何确定事务边界、如何开始和提交事务（这其实也就是定义了JTA，但JTA中的XA部分则是放在WS- Trsansaction中定义的）。同时考虑到Web服务的特性，在很多场景下传统的分布式事务（XA）是不能满足长时间运行的事务的需求的，因此 Web服务事务在WS-Transaction规范中提供了两类提交协议：WS-AT和WS-BA。前者是经典的分布式事务，后者则是为了长事务而提出 的，主要是利用了补偿的概念。  
  
这两个协议具体是如何运作的细节可以参考Web Service Transaction链接。该文章通过一个例子非常详细的描述了一个Web服务事务的运作过程，详细列出了其中交互的Web服务事务的SOAP消息，非常值得**学习** 。

**对分布式事务的理解**

## CAP定律

Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）



## Base理论

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：**即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性**。接下来看一下BASE中的三要素：

**1、基本可用**

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性----注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

**2、软状态**

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

**3、最终一致性**

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事物ACID特性是相反的，**它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态**。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

## 分布式事务

提到分布式系统，必然要提到分布式事务。要想理解分布式事务，不得不先介绍一下两阶段提交协议。先举个简单但不精准的例子来说明：

第一阶段，张老师作为“协调者”，给小强和小明（参与者、节点）发微信，组织他们俩明天8点在学校门口集合，一起去爬山，然后开始等待小强和小明答复。

第二阶段，如果小强和小明都回答没问题，那么大家如约而至。如果小强或者小明其中一人回答说“明天没空，不行”，那么张老师会立即通知小强和小明“爬山活动取消”。

细心的读者会发现，这个过程中可能有很多问题的。如果小强没看手机，那么张老师会一直等着答复，小明可能在家里把爬山装备都准备好了却一直等着张老师确认信息。更严重的是，如果到明天8点小强还没有答复，那么就算“超时”了，那小明到底去还是不去集合爬山呢？

这就是两阶段提交协议的弊病，所以后来业界又引入了三阶段提交协议来解决该类问题。

两阶段提交协议在主流开发语言平台，数据库产品中都有广泛应用和实现的，下面来介绍一下XOpen组织提供的DTP模型图：

XA协议指的是TM（事务管理器）和RM（资源管理器）之间的接口。目前主流的关系型数据库产品都是实现了XA接口的。JTA(Java Transaction API)是符合X/Open DTP模型的，事务管理器和资源管理器之间也使用了XA协议。 本质上也是借助两阶段提交协议来实现分布式事务的，下面分别来看看XA事务成功和失败的模型图：

在JavaEE平台下，WebLogic、Websphere等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的（其实笔者并不认为Tomcat能算是JavaEE应用服务器），这就需要借助第三方的框架Jotm、Atomikos等来实现，两者均支持spring事务整合。

而在Windows .NET平台中，则可以借助ado.net中的TransactionScop API来编程实现，还必须配置和借助Windows操作系统中的MSDTC服务。如果你的数据库使用的mysql，并且mysql是部署在Linux平台上的，那么是无法支持分布式事务的。 由于篇幅关系，这里不展开，感兴趣的读者可以自行查阅相关资料并实践。

总结：这种方式实现难度不算太高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况。但分布式事务对性能的影响会比较大，不适合高并发和高性能要求的场景。

## 提供回滚接口

在服务化架构中，功能X，需要去协调后端的A、B甚至更多的原子服务。那么问题来了，假如A和B其中一个调用失败了，那可怎么办呢？

在笔者的工作中经常遇到这类问题，往往提供了一个BFF层来协调调用A、B服务。如果有些是需要同步返回结果的，我会尽量按照“串行”的方式去调用。如果调用A失败，则不会盲目去调用B。如果调用A成功，而调用B失败，会尝试去回滚刚刚对A的调用操作。

当然，有些时候我们不必严格提供单独对应的回滚接口，可以通过传递参数巧妙的实现。

这样的情况，我们会尽量把可提供回滚接口的服务放在前面。举个例子说明：

我们的某个论坛网站，每天登录成功后会奖励用户5个积分，但是积分和用户又是两套独立的子系统服务，对应不同的DB，这控制起来就比较麻烦了。解决思路：

1. 把登录和加积分的服务调用放在BFF层一个本地方法中。
2. 当用户请求登录接口时，先执行加积分操作，加分成功后再执行登录操作
3. 如果登录成功，那当然最好了，积分也加成功了。如果登录失败，则调用加积分对应的回滚接口（执行减积分的操作）。

总结：这种方式缺点比较多，通常在复杂场景下是不推荐使用的，除非是非常简单的场景，非常容易提供回滚，而且依赖的服务也非常少的情况。

这种实现方式会造成代码量庞大，耦合性高。而且非常有局限性，因为有很多的业务是无法很简单的实现回滚的，如果串行的服务很多，回滚的成本实在太高。

## 本地消息表

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步伪代码如下，扣款1W，通过本地事务保证了凭证消息插入到消息表中。

第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

1. 采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件
2. 采用定时轮询扫描的方式，去检查消息表的数据。

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

## MQ（非事务消息）

通常情况下，在使用非事务消息支持的MQ产品时，我们很难将业务操作与对MQ的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。

先从消息生产者这端来分析，请看伪代码：

根据上述代码及注释，我们来分析下可能的情况：

1. 操作数据库成功，向MQ中投递消息也成功，皆大欢喜
2. 操作数据库失败，不会向MQ中投递消息了
3. 操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

1. 消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致
2. 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果

如何保证消息与业务操作一致，不丢失？

主流的MQ产品都具有持久化消息的功能。如果消费者宕机或者消费失败，都可以执行重试机制的（有些MQ可以自定义重试次数）。

如何避免消息被重复消费造成的问题？

1. 保证消费者调用业务的服务接口的幂等性
2. 通过消费日志或者类似状态表来记录消费状态，便于判断（建议在业务上自行实现，而不依赖MQ产品提供该特性）

总结：这种方式比较常见，性能和吞吐量是优于使用关系型数据库消息表的方案。如果MQ自身和业务都具有高可用性，理论上是可以满足大部分的业务场景的。不过在没有充分测试的情况下，不建议在交易业务中直接使用。

## MQ（事务消息）

举个例子，Bob向Smith转账，那我们到底是先发送消息，还是先执行扣款操作？

好像都可能会出问题。如果先发消息，扣款操作失败，那么Smith的账户里面会多出一笔钱。反过来，如果先执行扣款操作，后发送消息，那有可能扣款成功了但是消息没发出去，Smith收不到钱。除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

下面以阿里巴巴的RocketMQ中间件为例，分析下其设计和实现思路。

RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。细心的读者可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。如下图：

总结：据笔者的了解，各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

## 其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有bug或者程序逻辑有问题，那么重试1W次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

## 小结

上诉的几种方案中，笔者也大致总结了其设计思路，优势，劣势等，相信读者已经有了一定的理解。其实分布式系统的事务一致性本身是一个技术难题，目前没有一种很简单很完美的方案能够应对所有场景。具体还是要使用者根据不同的业务场景去抉择。

# 知识点回顾

## 分布式事务产生的原因

* 1. 数据库分库分表
  2. 服务soa化
  3. *在分布式系统中，每一个机器节点虽然都能明确的知道自己执行的事务是成功还是失败，但是却无法知道其他分布式节点的事务执行情况。因此，当一个事务要跨越多个分布式节点的时候（比如，下单流程，下单系统和库存系统可能就是分别部署在不同的分布式节点中），为了保证该事务可以满足ACID，就要引入一个协调者（Cooradinator）。其他的节点被称为参与者（Participant）。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务进行提交。*

# X/OPEN DTP

AP application， 应用程序

RM resource manager ，资源管理，一般表示数据库，必须实现XA定义的接口

TM transaction manager 事务管理器，负责协调和事务管理

## 2pc提交协议

1. 提交事务请求（投票）
2. 执行事务请求（提交或中断）

## 2pc的数据一致性问题

1. 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。
2. 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态
3. 二阶段无法解决的问题：协调者在发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交
4. 单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去

## 3pc提交协议

1. canCommit
2. preCommit
3. doCommit

改进点

1. 增加了超时机制
2. 第二阶段，如果协调者超时没有接受到参与者的反馈，则自动认为失败，发送abort命令
3. 第三阶段，如果参与者超时没有接受到协调者的反馈，则自动认为成功开始提交事务（基于概率）

## 3pc的问题

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

# XA/JTA

XA 就是 X/Open DTP 定义的事务管理器与资源管理器的接口规范（即接口函数），XA 接口函数由数据库厂商提供。

JTA是基于X/Open DTP模型开发的java transaction APi规范

# 概述

通过2pc的方式去完成分布式事务，虽然通过这种方式能够达到预期的效果，但是我们在现实中很少会用到2pc方式的提交的XA事务，有几个原因

1. 互联网电商应用的快速发展，对事务和数据的绝对一致性要求并没有传统企业应用那么高
2. XA事务的介入增加了TM中间件，使得系统复杂化
3. XA事务的性能不高，因为TM要等待RM回应，所以为了确保事务尽量成功提交，等待超时的时间通常比较长，比如30s到几分钟，如果RM出现故障或者响应比较慢，则整个事务的性能严重下降

# 互联网的分布式事务解决方案

目前互联网领域里有几种流行的分布式解决方案，但都没有像之前所说的XA事务一样形成X/OpenDTP那样的工业规范，而是仅仅在具体的行业里获得较多的认可

# 业务接口整合，避免分布式事务

这个方案就是把一个业务流程中需要在一个事务里执行的多个相关业务接口包装整合到一个事务中，比如我们可以讲A/B/C整合为一个服务D来实现单一事务的业务流程服务

# 最终一致性方案之ebay模式

eBay在2008年公布了一个关于BASE准则提到一个分布式事务解决方案。eBay的方案其实是一个最终一致性方案，它主要采用消息队列来辅助实现事务控制流程，方案的核心是将需要分布式处理的任务通过消息队列的方式来异步执行，如果事务失败，则可以发起人工重试的纠正流程。人工重试被更多的应用于支付场景，通过对账系统对事后问题进行处理

比如一个很常见的场景：某个用户产生了一笔交易，那么需要在交易表中增加记录，同时需要修改用户表的金额（余额），由于这两个表属于不同的远程服务，所以就会涉及到分布式事务与数据一致性的问题

|  |  |
| --- | --- |
| user(id, name, amt\_sold, amt\_bought)  transaction(xid, seller\_id, buyer\_id, amount) | begin;  INSERT INTO transaction VALUES(xid, $seller\_id, $buyer\_id, $amount);  UPDATE user SET amt\_sold = amt\_sold + $amount WHERE id = $seller\_id;  UPDATE user SET amt\_bought = amt\_bought + $amount WHERE id = $buyer\_id;  commit; |

那么在这里可以使用消息队列（MQ）来做

先启动一个事务，更新交易表（transaction）后，并不直接更新user表，而是将要对user表进行的更新插入到消息队列中。

目标系统收到该消息以后，启动本地事务去对用户表的余额做调整

伪代码

bool result=dao.update();

if(result){

mq.send();

}

根据上面的伪代码的实现方案，可能出现几种情况

1. 数据库操作成功，向MQ中投递消息也成功
2. 操作数据库失败，不会向MQ中投递消息
3. 操作数据库成功，但是向MQ中投递消息时失败，向外抛出异常。数据库操作回滚

对于上面几种情况，问题都不大。那么我们分析小消费端的问题

1. 消息出队列以后，消费者对应的业务操作要执行成功。如果执行失败，消息不能失效或者丢失。需要保证消息和业务操作一致
2. 尽量避免消息重复消费，如果重复消费，也不能影响业务的执行结果

对于第一个问题，如何保证消息不丢失

现在用的比较普遍的MQ都具有持久化消息的功能，如果消费者宕机或者消费失败，都可以执行重试机制

对于如何避免消息的重复消费

1. ***保证消费者的幂等性***；**也就是说如果队列中的消息因为网络异常导致发送多次的情况下，仍然需要保证消息被应用多次与应用一次产生的效果是一样的**
2. 通过消费日志表来记录消费状态；**增加一个message\_applied（msg\_id）表，用来记录已经被成功应用的消息。在目标系统执行更新操作之前，先检测该消息是否已经被消费过，消费完成后通过本地事务控制来更新这个“消费表状态”，用来避免消息重复消费问题**

上面这种方式是非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。

*各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大*

## 保证最终一致性的模式

1. 查询模式

任何一个服务操作都提供一个查询接口，用来向外部输出操作执行的状态。服务操作的使用方可以通过接口得知服务操作执行的状态，然后根据不同状态做不同的处理操作

为了能够实现查询，每个服务操作都需要有唯一的流水号

1. 补偿模式

有了查询模式，我们就能够得知操作所处的具体状态，如果整个操作处于不正常状态，我们需要修正操作中的出现问题的子操作。也许是要重新执行，或者取消已完成的操作。通过修复使得整个分布式系统达到最终一致。这个过程就是补偿模式

根据发起形式又分为

自动恢复：通过对发生失败操作的接口自动重试或者回滚已经完成的操作

通知运营：如果程序无法自动完成恢复，则通过运营人员手动进行补偿

通知技术：通过监控或者告警通知到技术人员，通过技术手段进行修复

# X/OpenDTP模型的支付宝的DTS架构

DTS(Distributed Transaction Service)框架是由支付宝在X/OpenDTP模型的基础上改进的一个设计，定义了类似2PC的标准两阶段接口，业务系统只需要实现对应的接口就可以使用DTS的事务功能。DTS最大的特点是放宽了数据库的强一致约束，保证了数据的最终一致性。

具体的流程是

|  |  |
| --- | --- |
|  | TCC分为三个阶段TRYING-CONFIRMING-CANCELING。每个阶段做不同的处理。  TRYING、CONFIRMING、CANCELIING大致可以理解为SQL事务中的LOCK、COMMIT、ROLLBACK  TRYING 阶段主要是对业务系统做检测及资源预留  CONFIRMING 阶段主要是对业务系统做确认提交，TRYING阶段执行成功并开始执行CONFIRMING阶段时，默认 CONFIRMING阶段是不会出错的。即：只要TRYING成功，CONFIRMING一定成功。  CANCELING 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。  以上所有的操作需要满足幂等性，幂等性的实现方式可以是：  1、通过唯一键值做处理，即每次调用的时候传入唯一键值，通过唯一键值判断业务是否被操作，如果已被操作，则不再重复操作  2、通过状态机处理，给业务数据设置状态，通过业务状态判断是否需要重复执行 |

如何更通俗的理解TCC事务模型

|  |
| --- |
| *支付系统接收到会员的支付请求后，需要扣减会员账户余额、增加会员积分（暂时假设需要同步实现）增加商户账户余额*  *会员系统、商户系统、积分系统是独立的三个子系统，无法通过传统的事务方式进行处理。*  *TRYING阶段：我们需要做的就是会员资金账户的资金预留，即：冻结会员账户的金额（订单金额）*  *CONFIRMING阶段：我们需要做的就是会员积分账户增加积分余额，商户账户增加账户余额*  *CANCELING阶段：该阶段需要执行的就是解冻释放我们扣减的会员余额* |

开源的tcc框架

tcc-transaction

bytetcc

# 最大努力通知型

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。